12 Kernel Methods

12.1 Introduction

In this chapter we continue our discussion of fixed-shape universal approximators, which began back in Section 11.2.3. This will very quickly lead to the notion of *kernelization* as a clever way of representing fixed-shape features so that they scale more gracefully when applied to vector-valued input.

12.2 Fixed-Shape Universal Approximators

Using the classic polynomial as our exemplar, in Section 11.2.3 we introduced the family of fixed-shape universal approximators as collections of various nonlinear functions which have no internal (tunable) parameters. In this section we pick up the discussion of fixed-shape approximators, beginning a deeper dive into the technicalities associated with these universal approximators and challenges we have to address when employing them in practice.

12.2.1 Trigonometric universal approximators

What generally characterizes fixed-shape universal approximators are their lack of internal parameters and straightforward organization, with very often the units of a particular family of fixed-shape approximators being organized in terms of degree or some other natural index (see Section 11.2.3). These simple characteristics have made fixed-shape approximators, such as the polynomials (which we have seen previously) as well as the sinusoidal and Fourier examples (which we discuss now) extremely popular in areas adjacent to machine learning, e.g., mathematics, physics, and engineering.

Example 12.1 Sinusoidal approximators

The collection of sine waves of increasing frequency is a common example of a classic fixed-shape approximator, with units of the form

$$f_1(x) = \sin(x), \quad f_2(x) = \sin(2x), \quad f_3(x) = \sin(3x), \quad \text{etc.}, \tag{12.1}$$

where the mth element in general is given as $f_m(x) = \sin(mx)$. The first four members of this family of functions are plotted in Figure 12.1.

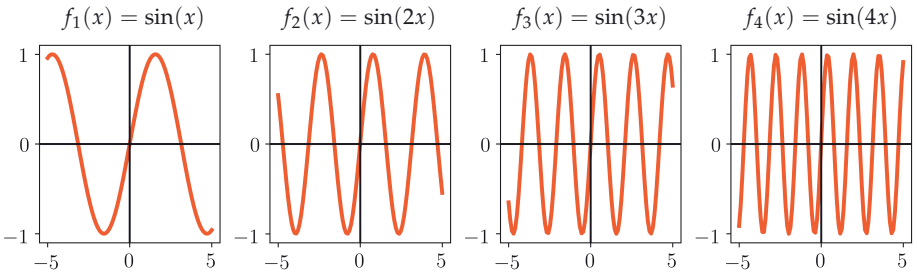

Figure 12.1 Figure associated with Example 12.1. From left to right, the first four units of a sinusoidal universal approximator. See text for further details.

As with polynomials, notice how each of these elements has no *tunable* parameter inside (and thus a *fixed* shape), and that the elements are naturally ordered in terms of their complexity from low to high. Also as with polynomials, we can easily generalize this catalog of functions to higher-dimensional input. In general, for N-dimensional input a sinusoidal unit takes the form

$$f_m(\mathbf{x}) = \sin(m_1 x_1)\sin(m_2 x_2)\cdots\sin(m_N x_N) \tag{12.2}$$

where $m_1, m_2, \ldots, m_N$ are nonnegative integers.

Example 12.2 The Fourier basis

Similarly to the sinusoidal family described in Example 12.1, the Fourier basis [56] – so named after its inventor Joseph Fourier who first used these functions in the early 1800s to study heat diffusion – consists of the set of paired sine and cosine waves (with ever-increasing frequency) of the form

$$f_{2m-1}(x) = \sin(2\pi m x) \quad \text{and} \quad f_{2m}(x) = \cos(2\pi m x) \tag{12.3}$$

for all $m \geq 1$. It is also common to write the Fourier units in the compact complex exponential form (see Exercise 12.1)

$$f_m(x) = e^{2\pi i m x}. \tag{12.4}$$

For a general N-dimensional input each multi-dimensional Fourier unit then takes the form

$$f_m(\mathbf{x}) = e^{2\pi i m_1 x_1} e^{2\pi i m_2 x_2} \cdots e^{2\pi i m_N x_N} \tag{12.5}$$

where $m_1, m_2, ..., m_N$ are integers.

12.2.2 The scaling challenge of fixed-shape approximators with large input dimension

As we saw in Chapter 11, when using polynomial units very often we use complete packages of monomials as a polynomial of a certain degree. For example, a polynomial model of degree D when the input is two-dimensional consists of all monomial units of the form $f_m(\mathbf{x}) = x_1^p x_2^q$ where p and q are nonnegative integers such that $0 < p + q \leq D$.

More generally with N-dimensional input a polynomial unit takes the form

$$f_m(\mathbf{x}) = x_1^{m_1} x_2^{m_2} \cdots x_N^{m_N} \tag{12.6}$$

and to construct a polynomial model of degree D we collect all such terms where $0 < m_1 + m_2 + \cdots + m_N \leq D$ and m_1, m_2, ..., m_N are nonnegative integers. Unless used in tandem with boosting (see Section 11.3) we virtually always use polynomial units as a *complete package* of units of a certain degree, not *individually*. One reason for doing this is that since the polynomial units are naturally ordered (from low to high complexity), when including a unit of a particular complexity it makes sense, organizationally speaking, to include all other units in the family having lesser complexities. For instance, it usually does not make much sense to define quadratic polynomials free of linear terms. Nonetheless, packaging polynomial units in this way is not something that we *must* do when employing them, but is a sensible and common practice.

Like polynomials, it is also common when employing sinusoidal and Fourier units to use them as complete packages, since they too are ordered in terms of their individual complexities from low to high. For example, in analogy to a degree-D polynomial, we can package a degree-D Fourier model consisting of *all units* of the form given in Equation (12.5) where $0 < \max(|m_1|, |m_2|, \ldots, |m_N|) \leq D$. This choice of packaging is largely a convention.

However, a very serious practical issue presents itself when employing fixed-shape approximators like polynomials and trigonometric bases, when used in complete packages of units: even with a moderate-sized input dimension N the corresponding number of units in the package M grows rapidly with N, quickly becoming prohibitively large in terms of storage and computation. In other words, the number of units of a typical fixed-shape approximator in a model employing a complete package of such units *grows exponentially* with the dimension of input.

Example 12.3 Number of units in a degree-D polynomial approximator

The precise number M of units in a degree-D polynomial of an input with dimension N can be computed precisely as

$$M = \begin{pmatrix} N+D \\ D \end{pmatrix} - 1 = \frac{(N+D)!}{N!D!} - 1. \tag{12.7}$$

Even if the input dimension N is of small to moderate size, e.g., $N = 100$ or $N = 1000$, then just the associated degree $D = 5$ polynomial feature map of these input dimensions has $M = 96,560,645$ and $M = 8,459,043,543,950$ monomial terms, respectively. In the latter case we cannot even hold the feature vectors in memory on a modern computer.

Example 12.4 Number of units in a degree-D Fourier approximator
The corresponding number of units M in a package of degree-D Fourier basis elements is even more gargantuan than that of a degree-D polynomial: the degree-D Fourier feature collection of arbitrary input dimension N has precisely

$$M = (2D+1)^N - 1 \tag{12.8}$$

units. When $D = 5$ and $N = 80$ this is $11^{80} - 1$, a number larger than current estimates of the number of atoms in the visible universe!

Our cursory analyses in Examples 12.3 and 12.4 indicate that since the total number of units of a fixed-shape approximator grows *exponentially* in the input dimension, any approach to selecting fixed-shape units for a nonlinear model is problematic in general. For example, with polynomials, even if we chose a smaller set of just those units with the exact same degree, i.e., all units where $m_1 + m_2 + \cdots + m_N = D$, we would still end up with a combinatorially large number of units to employ.

This serious scaling issue motivates the so-called *kernel trick* described in the next section, that extends the use of classic fixed-shape approximators (when employing complete packages of units) to problems with high-dimensional input.

12.3 The Kernel Trick

This crucial issue, of not being able to effectively store and compute with high-dimensional fixed-shape feature transformations, motivates the search for more efficient representations. In this section we introduce the notion of *kernelization*, also commonly called *the kernel trick*, as a clever way of constructing fixed-shape features for virtually any machine learning problem. Kernelization not only allows us to avoid the combinatorial explosion problem detailed at the end of the previous section, but also provides a way of generating new fixed-shape features defined solely through such a kernelized representation.

12.3.1 A useful fact from the fundamental theorem of linear algebra

Before discussing the concept of kernelization, it will be helpful to first recall a useful proposition from the fundamental theorem of linear algebra about decomposition of any M-dimensional vector $\boldsymbol{\omega}$ over the columns of a given $M \times P$ matrix $\mathbf{F}$. Denoting the pth column of $\mathbf{F}$ as $\mathbf{f}_p$, in the case where $\boldsymbol{\omega}$ happens to lie *inside* the column space of $\mathbf{F}$, we can express it via a linear combination of these columns as

$$\boldsymbol{\omega} = \sum_{p=1}^{P} \mathbf{f}_p z_p \tag{12.9}$$

where z_p is the linear combination weight or coefficient associated with $\mathbf{f}_p$. By stacking these weights into a $P \times 1$ column vector $\mathbf{z}$, we can write this relationship more compactly as

$$\boldsymbol{\omega} = \mathbf{F}\mathbf{z}. \tag{12.10}$$

If, on the other hand, $\boldsymbol{\omega}$ happens to lie *outside* the column space of $\mathbf{F}$, as illustrated pictorially in Figure 12.2, we can decompose it into two pieces – the portion of $\boldsymbol{\omega}$ belonging to the subspace spanned by the columns of $\mathbf{F}$, and an orthogonal component $\mathbf{r}$ – and write it as

$$\boldsymbol{\omega} = \mathbf{F}\mathbf{z} + \mathbf{r}. \tag{12.11}$$

Note that $\mathbf{r}$ being orthogonal to the span of columns in $\mathbf{F}$ means algebraically that $\mathbf{F}^T\mathbf{r} = \mathbf{0}_{P\times 1}$. Moreover when $\boldsymbol{\omega}$ is *in* the column space of $\mathbf{F}$, we can still decompose it using the more general form given in Equation (12.11) by setting $\mathbf{r} = \mathbf{0}_{M\times 1}$ without violating the orthogonality condition $\mathbf{F}^T\mathbf{r} = \mathbf{0}_{P\times 1}$.

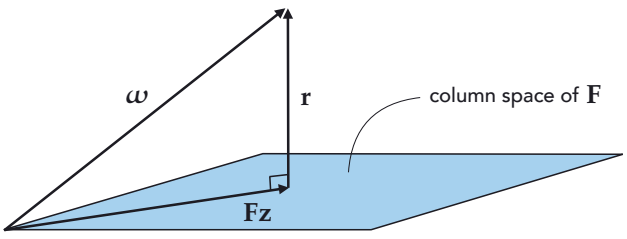

Figure 12.2 An illustration of a useful fact from the fundamental theorem of linear algebra which states that any vector $\boldsymbol{\omega}$ in an M-dimensional space can be decomposed as $\boldsymbol{\omega} = \mathbf{F}\mathbf{z} + \mathbf{r}$ where the vector $\mathbf{F}\mathbf{z}$ belongs in the column space of the matrix $\mathbf{F}$, and $\mathbf{r}$ is orthogonal to this subspace.

In sum, any vector $\boldsymbol{\omega}$ in an M-dimensional space can be decomposed over the column space of a given matrix $\mathbf{F}$ as $\boldsymbol{\omega} = \mathbf{F}\mathbf{z} + \mathbf{r}$. The vector $\mathbf{F}\mathbf{z}$ belongs in the

subspace determined by the columns of $\mathbf{F}$, while $\mathbf{r}$ is orthogonal to this subspace. As we will now see this simple decomposition is the key to representing fixed-shape features more effectively.

12.3.2 Kernelizing machine learning cost functions

Here we provide several fundamental examples of how to kernelize standard supervised machine learning problems and their cost functions, including the Least Squares cost for regression and the Softmax cost for two-class classification. Virtually all machine learning cost functions can be kernelized following arguments similar to these, including the multi-class Softmax, Principal Component Analysis, and K-means clustering (see chapter's exercises).

Example 12.5 Kernelizing regression via the Least Squares cost

Suppose we want to perform a generic nonlinear regression using our M units belonging to a degree-D fixed-shape approximator, with our corresponding model evaluated at the pth input $\mathbf{x}_p$ taking the form

$$\text{model}\left(\mathbf{x}_p, \mathbf{w}\right) = w_0 + f_1\left(\mathbf{x}_p\right)w_1 + f_2\left(\mathbf{x}_p\right)w_2 + \cdots + f_M\left(\mathbf{x}_p\right)w_M. \tag{12.12}$$

For convenience we will write this more compactly, exposing the feature-touching weights and the bias separately, as

$$\text{model}\left(\mathbf{x}_p, b, \boldsymbol{\omega}\right) = b + \mathbf{f}_p^T\boldsymbol{\omega} \tag{12.13}$$

where we have used the bias/feature-touching weight notation (previously introduced in, e.g., Section 6.4.5)

$$b = w_0 \qquad \text{and} \qquad \boldsymbol{\omega} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_M \end{bmatrix} \tag{12.14}$$

as well as a shorthand for our entire set of M feature transformations of the training input $\mathbf{x}_p$ as

$$\mathbf{f}_p = \begin{bmatrix} f_1(\mathbf{x}_p) \\ f_2(\mathbf{x}_p) \\ \vdots \\ f_M(\mathbf{x}_p) \end{bmatrix}. \tag{12.15}$$

In this notation our Least Squares cost for regression takes the form

$$g\left(b, \boldsymbol{\omega}\right) = \frac{1}{P}\sum_{p=1}^{P}\left(b + \mathbf{f}_p^T\boldsymbol{\omega} - y_p\right)^2. \tag{12.16}$$

Now, denote by $\mathbf{F}$ the $M \times P$ matrix formed by stacking the vectors $\mathbf{f}_p$ column-wise. Employing the fundamental theorem of linear algebra discussed in the previous section, we may write $\boldsymbol{\omega}$ as

$$\boldsymbol{\omega} = \mathbf{F}\mathbf{z} + \mathbf{r} \tag{12.17}$$

where $\mathbf{r}$ satisfies $\mathbf{F}^T\mathbf{r} = \mathbf{0}_{P\times 1}$. Substituting this representation of $\boldsymbol{\omega}$ back into the cost function in Equation (12.16) gives

$$\frac{1}{P}\sum_{p=1}^{P}\left(b + \mathbf{f}_p^T\left(\mathbf{F}\mathbf{z} + \mathbf{r}\right) - y_p\right)^2 = \frac{1}{P}\sum_{p=1}^{P}\left(b + \mathbf{f}_p^T\mathbf{F}\mathbf{z} - y_p\right)^2. \tag{12.18}$$

Finally, denoting the symmetric $P \times P$ matrix $\mathbf{H} = \mathbf{F}^T\mathbf{F}$ (and its pth column $\mathbf{h}_p = \mathbf{F}^T\mathbf{f}_p$), referred to as a *kernel matrix* or just a *kernel* for short, our original cost function can be expressed equivalently as

$$g\left(b, \mathbf{z}\right) = \frac{1}{P}\sum_{p=1}^{P}\left(b + \mathbf{h}_p^T\mathbf{z} - y_p\right)^2 \tag{12.19}$$

with our corresponding model evaluated at the pth input now taking the equivalent form

$$\text{model}\left(\mathbf{x}_p, b, \mathbf{z}\right) = b + \mathbf{h}_p^T\mathbf{z}. \tag{12.20}$$

Note that in kernelizing the original regression model in Equation (12.13) and its associated cost function in Equation (12.16) we have changed their arguments (due to our substitution of $\boldsymbol{\omega}$), arriving at completely equivalent *kernelized* model in Equation (12.20) and *kernelized* cost function in Equation (12.19).

Example 12.6 Kernelizing two-class classification via the Softmax cost
Following the pattern shown in Example 12.5, here we essentially repeat the same argument employing the two-class Softmax cost.

Writing our generic two-class Softmax cost using the same notation as employed in Example 12.5 we have

$$g\left(b, \boldsymbol{\omega}\right) = \frac{1}{P}\sum_{p=1}^{P}\log\left(1 + e^{-y_p\left(b + \mathbf{f}_p^T\boldsymbol{\omega}\right)}\right). \tag{12.21}$$

We then write the representation of $\boldsymbol{\omega}$ over $\mathbf{F}$ as $\boldsymbol{\omega} = \mathbf{F}\mathbf{z} + \mathbf{r}$ where $\mathbf{F}^T\mathbf{r} = \mathbf{0}_{P\times 1}$. Making this substitution into Equation (12.21) and simplifying gives

$$g\left(b, \mathbf{z}\right) = \frac{1}{P}\sum_{p=1}^{P}\log\left(1 + e^{-y_p\left(b + \mathbf{f}_p^T\mathbf{F}\mathbf{z}\right)}\right). \tag{12.22}$$

Denoting the $P \times P$ kernel matrix $\mathbf{H} = \mathbf{F}^T\mathbf{F}$ (where $\mathbf{h}_p = \mathbf{F}^T\mathbf{f}_p$ is the pth column of $\mathbf{H}$) we can then write the cost function in Equation (12.22) in *kernelized* form as

$$g\left(b, \mathbf{z}\right) = \frac{1}{P}\sum_{p=1}^{P}\log\left(1 + e^{-y_p\left(b + \mathbf{h}_p^T\mathbf{z}\right)}\right). \tag{12.23}$$

This kernelized form of the two-class Softmax is often referred to as *kernelized logistic regression*.

Using the same sort of argument given in Examples 12.5 and 12.6 we may kernelize virtually any machine learning problem discussed in this text including multi-class classification, Principal Component Analysis, K-means clustering, as well as any ℓ_2 regularized version of these models. For easy reference, we show both the original and kernelized forms of popular supervised learning cost functions in Table 12.1.

Table 12.1 Popular supervised learning cost functions and their kernelized versions.

Cost function	Original version	Kernelized version
Least Squares	$\frac{1}{P}\sum_{p=1}^{P}\left(b + \mathbf{f}_p^T\boldsymbol{\omega} - y_p\right)^2$	$\frac{1}{P}\sum_{p=1}^{P}\left(b + \mathbf{h}_p^T\mathbf{z} - y_p\right)^2$
Two-class Softmax	$\frac{1}{P}\sum_{p=1}^{P}\log\left(1 + e^{-y_p\left(b + \mathbf{f}_p^T\boldsymbol{\omega}\right)}\right)$	$\frac{1}{P}\sum_{p=1}^{P}\log\left(1 + e^{-y_p\left(b + \mathbf{h}_p^T\mathbf{z}\right)}\right)$
Squared-margin SVM	$\frac{1}{P}\sum_{p=1}^{P}\max^2\left(0, 1 - y_p\left(b + \mathbf{f}_p^T\boldsymbol{\omega}\right)\right)$	$\frac{1}{P}\sum_{p=1}^{P}\max^2\left(0, 1 - y_p\left(b + \mathbf{h}_p^T\mathbf{z}\right)\right)$
Multi-class Softmax	$\frac{1}{P}\sum_{p=1}^{P}\log\left(1 + \sum_{\substack{j=0\\ j\neq y_p}}^{C-1} e^{\left(b_j - b_{y_p}\right) + \mathbf{f}_p^T\left(\boldsymbol{\omega}_j - \boldsymbol{\omega}_{y_p}\right)}\right)$	$\frac{1}{P}\sum_{p=1}^{P}\log\left(1 + \sum_{\substack{j=0\\ j\neq y_p}}^{C-1} e^{\left(b_j - b_{y_p}\right) + \mathbf{h}_p^T\left(\mathbf{z}_j - \mathbf{z}_{y_p}\right)}\right)$
ℓ_2 regularizer [a]	$\lambda\left\|\boldsymbol{\omega}\right\|_2^2$	$\lambda\,\mathbf{z}^T\mathbf{H}\mathbf{z}$

[a] The ℓ_2 regularizer can be added to any cost function $g(b, \boldsymbol{\omega})$ in the middle column and the resulting kernelized form of the sum $g(b, \boldsymbol{\omega}) + \lambda\left\|\boldsymbol{\omega}\right\|_2^2$ will be the sum of the kernelized cost and the kernelized regularizer, i.e., $g(b, \mathbf{z}) + \lambda\,\mathbf{z}^T\mathbf{H}\mathbf{z}$.

12.3.3 Popular kernels in machine learning

The real value of kernelizing any machine learning cost is that for many fixed-shape units, including polynomial and Fourier features, the kernel matrix $\mathbf{H} = \mathbf{F}^T\mathbf{F}$ may be constructed *without* first building the matrix $\mathbf{F}$ (which often has prohibitively large row dimension). Instead, as we will see through a number of examples, this matrix may be constructed *entry-wise* via simple formulae. Moreover, thinking about constructing kernel matrices in this way leads to the construction of fixed-shape universal approximators starting with the definition of the kernel matrix itself (and not by beginning with an explicit feature transformation). In either case, by constructing the kernel matrix without first computing $\mathbf{F}$ we completely avoid the exponential scaling problem with fixed-shape universal approximators discussed in Section 12.2.2.

Example 12.7 The polynomial kernel

Consider the following degree $D = 2$ polynomial mapping from $N = 2$ to $M = 5$ dimensional space given by the feature transformation vector

$$\mathbf{f} = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_1 x_2 \\ x_2^2 \end{bmatrix}. \tag{12.24}$$

Note that multiplying some or all of the entries in $\mathbf{f}$ by a constant value like $\sqrt{2}$, as in

$$\mathbf{f} = \begin{bmatrix} \sqrt{2}\,x_1 \\ \sqrt{2}\,x_2 \\ x_1^2 \\ \sqrt{2}\,x_1 x_2 \\ x_2^2 \end{bmatrix} \tag{12.25}$$

does not change this feature transformation for our modeling purposes, since the $\sqrt{2}$ attached to several of the terms can be absorbed by their associated weights in $\boldsymbol{\omega}$ when forming model $(\mathbf{x}, b, \boldsymbol{\omega}) = b + \mathbf{f}^T\boldsymbol{\omega}$. Denoting briefly by $\mathbf{u} = \mathbf{x}_i$ and $\mathbf{v} = \mathbf{x}_j$ the ith and jth input data points, respectively, the (i, j)th element of the kernel matrix $\mathbf{H} = \mathbf{F}^T\mathbf{F}$ for a degree $D = 2$ polynomial is written as

$$
\begin{aligned}
h_{i,j} &= \mathbf{f}_i^T \mathbf{f}_j \\
&= \begin{bmatrix} \sqrt{2}\,u_1 & \sqrt{2}\,u_2 & u_1^2 & \sqrt{2}\,u_1 u_2 & u_2^2 \end{bmatrix} \begin{bmatrix} \sqrt{2}\,v_1 \\ \sqrt{2}\,v_2 \\ v_1^2 \\ \sqrt{2}\,v_1 v_2 \\ v_2^2 \end{bmatrix} \\
&= \left(1 + 2\,u_1 v_1 + 2\,u_2 v_2 + u_1^2 v_1^2 + 2\,u_1 u_2 v_1 v_2 + u_2^2 v_2^2\right) - 1 \\
&= \left(1 + u_1 v_1 + u_2 v_2\right)^2 - 1 \\
&= \left(1 + \mathbf{u}^T \mathbf{v}\right)^2 - 1.
\end{aligned}
\tag{12.26}
$$

In other words, the kernel matrix $\mathbf{H}$ may be built without first constructing the explicit features in Equation (12.25), by simply defining it entry-wise as

$$h_{i,j} = \left(1 + \mathbf{x}_i^T \mathbf{x}_j\right)^2 - 1. \tag{12.27}$$

This way of defining the polynomial kernel matrix is very useful since we only require access to the original input data, not the explicit polynomial features themselves.

Although the kernel construction rule in Equation (12.27) was derived specifically for $N = 2$ and a degree $D = 2$ polynomial, one can show that a polynomial kernel can be defined entry-wise, in a similar manner, for general N and D as

$$h_{i,j} = \left(1 + \mathbf{x}_i^T \mathbf{x}_j\right)^D - 1. \tag{12.28}$$

Example 12.8 The Fourier kernel

A degree-D Fourier feature transformation from $N = 1$ to $M = 2D$ dimensional space may be written as the $2D \times 1$ feature vector

$$\mathbf{f} = \begin{bmatrix} \sqrt{2}\cos\left(2\pi x\right) \\ \sqrt{2}\sin\left(2\pi x\right) \\ \vdots \\ \sqrt{2}\cos\left(2\pi D x\right) \\ \sqrt{2}\sin\left(2\pi D x\right) \end{bmatrix} \tag{12.29}$$

where, as explained in the previous example, the multiplication of its entries by $\sqrt{2}$ does not alter the original transformation defined in Equation (12.3) for our modeling purposes. In this case the corresponding (i, j)th element of the kernel matrix $\mathbf{H} = \mathbf{F}^T\mathbf{F}$ can be written as

$$h_{i,j} = \mathbf{f}_i^T \mathbf{f}_j = \sum_{m=1}^{D} 2\left[\cos(2\pi m x_i)\cos\left(2\pi m x_j\right) + \sin(2\pi m x_i)\sin\left(2\pi m x_j\right)\right]. \quad (12.30)$$

Using the simple trigonometric identity $\cos(\alpha)\cos(\beta) + \sin(\alpha)\sin(\beta) = \cos(\alpha - \beta)$, this may be written equivalently as

$$h_{i,j} = \sum_{m=1}^{D} 2\cos\left(2\pi m(x_i - x_j)\right). \quad (12.31)$$

Employing the complex definition of cosine, i.e., $\cos(\alpha) = \frac{e^{i\alpha}+e^{-i\alpha}}{2}$, we can rewrite this as

$$h_{i,j} = \sum_{m=1}^{D}\left[e^{2\pi i m(x_i - x_j)} + e^{-2\pi i m(x_i - x_j)}\right] = \left[\sum_{m=-D}^{D} e^{2\pi i m(x_i - x_j)}\right] - 1. \quad (12.32)$$

If $x_i - x_j$ is an integer then $e^{2\pi i m\left(x_i - x_j\right)} = 1$, and the summation expression inside brackets in Equation (12.32) sums to $2D + 1$. Supposing this is not the case, examining the summation alone we may write

$$\sum_{m=-D}^{D} e^{2\pi i m\left(x_i - x_j\right)} = e^{-2\pi i D\left(x_i - x_j\right)} \sum_{m=0}^{2D} e^{2\pi i m\left(x_i - x_j\right)}. \quad (12.33)$$

Noticing that the sum on the right-hand side is a geometric series, we can further simplify the above as

$$e^{-2\pi i D\left(x_i - x_j\right)} \frac{1 - e^{2\pi i\left(x_i - x_j\right)(2D+1)}}{1 - e^{2\pi i\left(x_i - x_j\right)}} = \frac{\sin\left((2D+1)\,\pi\left(x_i - x_j\right)\right)}{\sin\left(\pi\left(x_i - x_j\right)\right)} \quad (12.34)$$

where the final equality follows from the complex definition of sine, i.e., $\sin(\alpha) = \frac{e^{i\alpha}-e^{-i\alpha}}{2i}$.

Because in the limit, as t approaches any integer value, we have $\frac{\sin((2D+1)\pi t)}{\sin(\pi t)} = 2D + 1$ (which one can show using L'Hospital's rule from basic calculus), we may therefore generally write, in conclusion, that

$$h_{i,j} = \frac{\sin\left((2D+1)\,\pi\left(x_i - x_j\right)\right)}{\sin\left(\pi\left(x_i - x_j\right)\right)} - 1. \quad (12.35)$$

Similar Fourier kernel derivation can be made for a general N-dimensional input (see Exercise 12.11).

Example 12.9 The Radial Basis Function (RBF) kernel

Another popular choice of kernel is the Radial Basis Function (RBF) kernel defined entry-wise over the input data as

$$h_{i,j} = e^{-\beta \left\| \mathbf{x}_i - \mathbf{x}_j \right\|_2^2} \tag{12.36}$$

where $\beta > 0$ is a *hyperparameter* that must be tuned to the data. While the RBF kernel is typically defined directly as the kernel matrix in Equation (12.36), it can be traced back to an explicit feature transformation as with the polynomial and Fourier kernels. That is, we can find the explicit form of the fixed-shape feature transformation $\mathbf{f}$ such that

$$h_{i,j} = \mathbf{f}_i^T \mathbf{f}_j \tag{12.37}$$

where $\mathbf{f}_i$ and $\mathbf{f}_j$ are the feature transformations of the input points $\mathbf{x}_i$ and $\mathbf{x}_j$, respectively. The RBF feature transformation is different from polynomial and Fourier transformations in that its associated feature vector $\mathbf{f}$ is *infinite*-dimensional. For example, when $N = 1$ the feature vector $\mathbf{f}$ takes the form

$$\mathbf{f} = \begin{bmatrix} f_1(x) \\ f_2(x) \\ f_3(x) \\ \vdots \end{bmatrix} \tag{12.38}$$

where the mth entry (or feature) is defined as

$$f_m(x) = e^{-\beta x^2} \sqrt{\frac{(2\beta)^{m-1}}{(m-1)!}}\, x^{m-1} \quad \text{for all } m \geq 1. \tag{12.39}$$

When $N > 1$ the corresponding feature vector takes on an analogous form which is also infinite in length, making it impossible to even construct and store such a feature vector (regardless of the input dimension).

Notice that the shape (and hence fitting behavior) of RBF kernels depends on the setting of their hyperparameter β. In general, the larger β is set the more complex an associated model employing an RBF kernel becomes. To illustrate this, in Figure 12.3 we show three examples of supervised learning: regression (top row), two-class classification (middle row), and multi-class classification (bottom row), using the RBF kernel with three distinct settings of β in each instance. This creates underfitting (left column), reasonable predictive behavior (middle column), and overfitting behavior (right column). In each instance Newton's method was used to minimize each corresponding cost, and consequently tune each model's parameters. In practice β is set via *cross-validation* (see, e.g., Example 12.10).

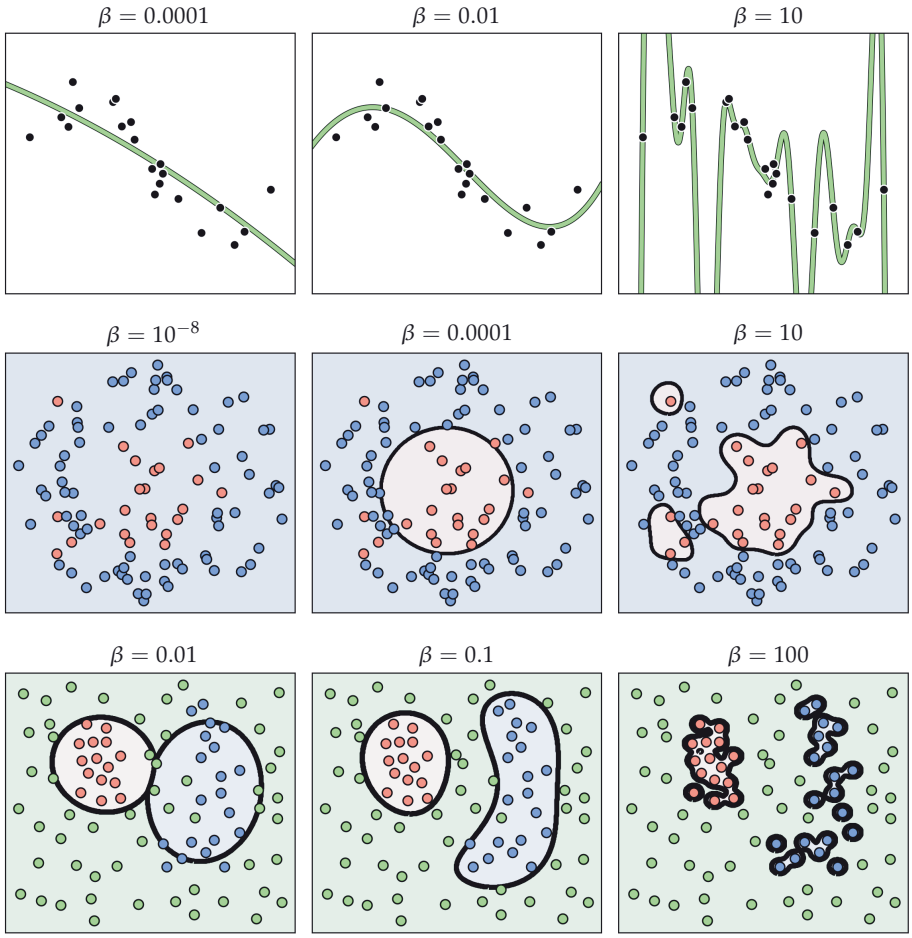

Figure 12.3 Figure associated with Example 12.9. See text for details.

While we have presented some of the most commonly used kernels in practice here, the reader can see, e.g., [57, 58] for a more exhaustive list of kernels and their properties.

12.3.4 Making predictions with kernelized models

As we saw in Examples 12.5 and 12.6, the kernelized form of a general supervised model evaluated at a point $\mathbf{x}$ takes the form

$$\text{model}\,(\mathbf{x}, b, \mathbf{z}) = b + \mathbf{h}^T\mathbf{z} \tag{12.40}$$

where the parameters b and $\mathbf{z}$ must be tuned by the minimization of an appropriate kernelized cost. In this framework the kernelization $\mathbf{h}$ of the generic

input $\mathbf{x}$ involves evaluation against *every* point $\mathbf{x}_p$ in the (training) dataset. For example with a degree-D polynomial kernel, $\mathbf{h}$ is given as the P-dimensional vector

$$\mathbf{h} = \begin{bmatrix} \left(1 + \mathbf{x}_1^T\mathbf{x}\right)^D - 1 \\ \left(1 + \mathbf{x}_2^T\mathbf{x}\right)^D - 1 \\ \vdots \\ \left(1 + \mathbf{x}_P^T\mathbf{x}\right)^D - 1 \end{bmatrix}. \tag{12.41}$$

This necessity of employing every (training) data point in evaluating a trained model is virtually unique[1] to kernelized learners, as we will not see this requirement when employing other universal approximators in the chapters to come.

12.4 Kernels as Measures of Similarity

If we look back at the form of the polynomial, Fourier, and RBF kernels in Examples 12.7 through 12.9 we can see that in each instance the (i, j)th entry of the kernel matrix is a function defined on the pair $\left(\mathbf{x}_i, \mathbf{x}_j\right)$ of input data. For example, studying the RBF kernel

$$h_{i,j} = e^{-\beta \left\|\mathbf{x}_i - \mathbf{x}_j\right\|_2^2} \tag{12.42}$$

we can see that, as a function of $\mathbf{x}_i$ and $\mathbf{x}_j$, it measures the similarity between these two inputs via the ℓ_2 norm of their difference. The more similar $\mathbf{x}_i$ and $\mathbf{x}_j$ are in the input space the larger $h_{i,j}$ becomes, and vice versa. In other words, the RBF kernel can be interpreted as a *similarity measure* that describes how closely two inputs resemble each other. This interpretation of kernels as similarity measures also applies to other previously introduced kernels including the polynomial and Fourier kernels, even though these kernels clearly encode similarity in different ways.

In Figure 12.4 we visualize our three exemplar kernels (polynomial, Fourier, and RBF) as similarity measures by fixing $\mathbf{x}_i$ at $\mathbf{x}_i = [0.5 \;\; 0.5]^T$ and plotting $h_{i,j}$ for a fine range of $\mathbf{x}_j$ values over the unit square $[0, 1]^2$, producing a color-coded surface showing how each kernel treats points near $\mathbf{x}_i$. Analyzing this figure we can obtain a general sense of how these three kernels define *similarity* between points. Firstly, we can see that a polynomial kernel treats data points $\mathbf{x}_i$ and $\mathbf{x}_j$ similarly if their inner product is high or, in other words, they highly correlate with each other. Likewise, the points are treated as dissimilar when they are

[1] The evaluation of a K-nearest-neighbors classifier also involves employing the entire training set.

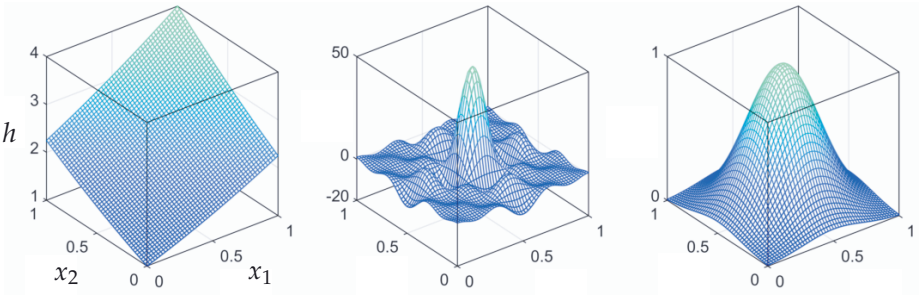

Figure 12.4 Surfaces generated by polynomial, Fourier, and RBF kernels centered at $\mathbf{x}_i = [0.5 \;\; 0.5]^T$. Each surface point is color-coded based on its magnitude, which can be thought of as a measure of similarity between $\mathbf{x}_i$ and its corresponding input. (left panel) A degree $D = 2$ polynomial kernel, (middle panel) degree $D = 3$ Fourier kernel, and (right panel) RBF kernel with $\beta = 10$. See text for further details.

orthogonal to one another. On the other hand, the Fourier kernel treats points as similar if they lie close together, but their similarity differs like a sinc function as their distance from each other grows. Finally, an RBF kernel provides a smooth similarity between points: if they are close to each other in a *Euclidean* sense they are deemed highly similar, but once the distance between them passes a certain threshold they become rapidly dissimilar.

12.5 Optimization of Kernelized Models

As discussed previously, virtually any machine learning model (supervised or unsupervised) can be kernelized. The real value in kernelization is that, for a large range of kernel types, we can actually construct the kernel matrix $\mathbf{H}$ *without* explicitly defining the associated feature transformations. As we have seen this allows us to get around the scaling issue associated with fixed-shape approximators with large input dimension (see Section 12.2.2). Moreover, because the final kernelized model remains linear in its parameters, corresponding kernelized cost functions are quite "nice" in terms of their general geometry. For example, any convex cost function for regression and classification *remains* convex when kernelized, including popular cost functions for regression, two-class, and multi-class classification (detailed in Chapters 5–7). This allows virtually any optimization method to be used to tune a kernelized supervised learner, from zero- to first-order and even powerful second-order approaches like Newton's method (detailed in Chapters 2–4).

However, because a generic kernel matrix $\mathbf{H}$ is a square matrix of size $P \times P$ (where P is the number of data points in the training set) kernelized models inherently scale quadratically (and thus very poorly) in the size of training data. This not only makes training kernelized models extremely challenging on large

datasets, but also predictions using such models (which as we saw in Section 12.3.4 require the evaluation of *every* training data point) become increasingly challenging as the size of training data increases.

Most standard ways of dealing with this crippling scaling issue in the size of the training data revolve around avoiding the creation of the entire kernel matrix $\mathbf{H}$ at once, especially during training. For example, one can use first-order methods such as stochastic gradient descent so that only a small number of training data points are dealt with at a time, meaning that only a small subset of columns of $\mathbf{H}$ are ever created concurrently when training. Sometimes the structure of certain problems can be used to avoid explicit kernel construction as well.

12.6 Cross-Validating Kernelized Learners

In general, there is a large difference between the *capacity* of subsequent degrees D and $D+1$ in models employing polynomial and Fourier kernels. With polynomials for instance, the difference between the number of units encapsulated in a degree-D polynomial kernel and that of a degree-$(D+1)$ polynomial kernel can be calculated, using Equation (12.7), as

$$\left[\begin{pmatrix} N+D+1 \\ D+1 \end{pmatrix} - 1\right] - \left[\begin{pmatrix} N+D \\ D \end{pmatrix} - 1\right] = \begin{pmatrix} N+D \\ D+1 \end{pmatrix}. \tag{12.43}$$

When $N = 500$, for example, there are $20,958,500$ more polynomial units encapsulated in a degree $D = 3$ kernel matrix than a degree $D = 2$ kernel matrix. Because of this enormous combinatorial leap in capacity between subsequent degree kernels, cross-validation via *regularization* with the ℓ_2 norm (as detailed in Section 11.4) is common practice when employing polynomial and Fourier kernels. Since the hyperparameter β of the RBF kernel is continuous, models employing an RBF kernel can (in addition to the regularization approach) be cross-validated in principle by comparing various values of β directly.

Example 12.10 Breast cancer classification using an RBF kernel

In this example we use naive cross-validation (as first detailed in Section 11.4.2) to determine an ideal parameter β for an RBF kernel over the breast cancer dataset first described in Exercise 6.13. In this set of experiments we use the Softmax cost and set aside 20 percent of this two-class dataset (randomly) for validation purposes (with the same portion being set aside for validation for each value of β used). We try out a range of 50 evenly spaced values for β on the interval $[0, 1]$, minimize the corresponding cost using Newton's method, and plot the number of misclassifications on both the training (in blue) and validation (in yellow) sets in Figure 12.5. The minimum number of misclassifications on the

validation set occurred when β was set close to the value 0.2, which resulted in one and five misclassifications on the training and validation sets, respectively. A simple linear classifier trained on the same portion of data provided seven and 22 misclassifications on training and validation sets, respectively.

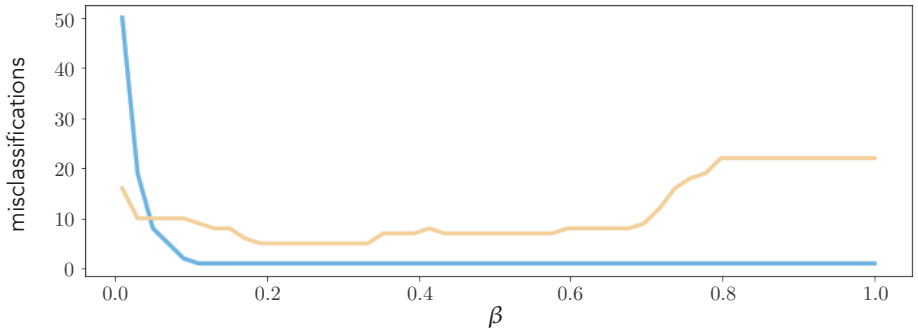

Figure 12.5 Figure associated with Example 12.10. See text for details.

12.7 Conclusion

In this chapter we continued our description of fixed-shape models, continuing from Section 11.2.3 where they were first introduced. We began with a review of several popular examples of fixed-shape universal approximators in Section 12.2. The fact that these universal approximators scale extremely poorly with the input dimension of a dataset, as described in Section 12.2.2, naturally led us to discuss their extension as *kernels* in Section 12.3. Using the "kernel trick" we can not only extend popular fixed-shape approximators to more easily deal with high-dimensional input, but can create a range of new approximators directly as kernels. However, while *kernelizing* a fixed-shape approximator helps it overcome scaling issues in the input dimension of a dataset, it introduces a scaling issue in the *dataset size*. This issue can be somewhat ameliorated via clever kernel matrix construction and optimization (as outlined in Section 12.5). Lastly, in Section 12.6 we briefly touched on the use of regularizer based cross-validation – which was previously discussed at length in Section 11.6.

12.8 Exercises

† The data required to complete the following exercises can be downloaded from the text's github repository at `github.com/jermwatt/machine_learning_refined`

12.1 Complex Fourier representation

Verify that using complex exponential definitions of cosine and sine functions, i.e., $\cos(\alpha) = \frac{1}{2}\left(e^{i\alpha} + e^{-i\alpha}\right)$ and $\sin(\alpha) = \frac{1}{2i}\left(e^{i\alpha} - e^{-i\alpha}\right)$, we can write the partial Fourier expansion model

$$\text{model}(x, \mathbf{w}) = w_0 + \sum_{m=1}^{M} \left[\cos(2\pi m x)\, w_{2m-1} + \sin(2\pi m x)\, w_{2m}\right] \tag{12.44}$$

equivalently as

$$\text{model}(x, \mathbf{v}) = \sum_{m=-M}^{M} e^{2\pi i m x}\, v_m \tag{12.45}$$

where the complex weights $v_{-M}, ..., v_0, ..., v_M$ are given in terms of the real weights $w_0, w_1, ..., w_{2M}$ as

$$v_m = \begin{cases} \frac{1}{2}\left(w_{2m-1} - i\, w_{2m}\right) & \text{if } m > 0 \\ w_0 & \text{if } m = 0 \\ \frac{1}{2}\left(w_{1-2m} + i\, w_{-2m}\right) & \text{if } m < 0. \end{cases} \tag{12.46}$$

12.2 Combinatorial explosion in monomials

Confirm that the number of monomial units in a degree-D polynomial grows combinatorially in input dimension as given in Equation (12.7).

12.3 Polynomial kernel regression

Reproduce polynomial kernel fits of degree $D = 1$, $D = 3$, and $D = 12$ to the nonlinear dataset shown in Figure 12.6.

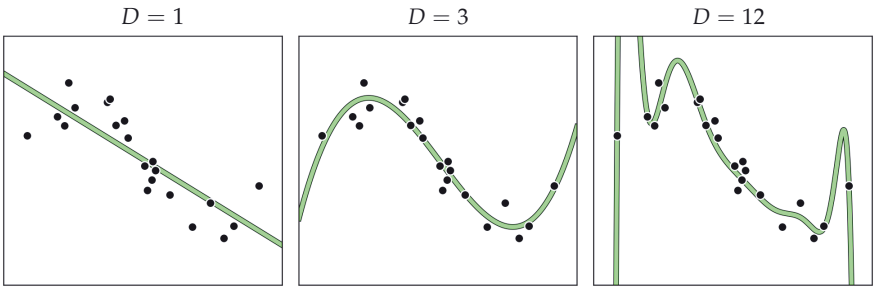

Figure 12.6 Figure associated with Exercise 12.3.

12.4 **Kernelize the ℓ_2 regularized Least Squares cost**

Use the kernelization argument made in Examples 12.5 and 12.6 to kernelize the ℓ_2 regularized Least Squares cost function.

12.5 **Kernelize the multi-class Softmax cost**

Use the kernelization argument made in Examples 12.5 and 12.6 to kernelize the multi-class Softmax cost.

12.6 **Regression with the RBF kernel**

Implement the RBF kernel in Example 12.9 and perform nonlinear regression on the dataset shown in the top row of Figure 12.3 using $\beta = 10^{-4}$, $\beta = 10^{-2}$, and $\beta = 10$ to reproduce the respective fits shown in the figure.

12.7 **Two-class classification with the RBF kernel**

Implement the RBF kernel in Example 12.9 and perform nonlinear two-class classification on the dataset shown in the middle row of Figure 12.3 using $\beta = 10^{-8}$, $\beta = 10^{-4}$, and $\beta = 10$. For each case produce a misclassification history plot to show that your results match what is shown in the figure.

12.8 **Multi-class classification with the RBF kernel**

Implement the RBF kernel in Example 12.9 and perform nonlinear multi-class classification on the dataset shown in the bottom row of Figure 12.3 using $\beta = 10^{-2}$, $\beta = 10^{-1}$, and $\beta = 100$. For each case produce a misclassification history plot to show that your results match, respectively, what is shown in the figure.

12.9 **Polynomial kernels for arbitrary degree and input dimension**

Show that a polynomial kernel can be defined entry-wise, as given in Equation (12.28), for general degree D and input dimension N.

12.10 **An infinite-dimensional feature transformation**

Verify that the infinite-dimensional feature transformation defined in Equation (12.39) indeed yields the entry-wise form of the RBF kernel in Equation (12.36).

12.11 **Fourier kernel for vector-valued input**

For a general N-dimensional input each Fourier unit takes the form

$$f_{\mathbf{m}}(\mathbf{x}) = e^{2\pi i m_1 x_1} e^{2\pi i m_2 x_2} \cdots e^{2\pi i m_N x_N} = e^{2\pi i \mathbf{m}^T \mathbf{x}} \tag{12.47}$$

where the vector $\mathbf{m}$

$$\mathbf{m} = \begin{bmatrix} m_1 \\ m_2 \\ \vdots \\ m_N \end{bmatrix} \tag{12.48}$$

contains integer-valued entries. Further, a degree-D Fourier expansion contains all such units satisfying $0 < \|\mathbf{m}\|_\infty \leq D$ (see Appendix Section C.5 if not familiar with the infinity norm). Calculate the corresponding (i, j)th entry of the kernel matrix $\mathbf{H}$, i.e., $h_{i,j} = \mathbf{f}_i^T \overline{\mathbf{f}_j}$ where $\overline{\mathbf{f}_j}$ denotes the complex conjugate of $\mathbf{f}_j$.

12.12 Kernels and a cancer dataset

Repeat the experiment described in Example 12.10, and produce a plot like the one shown in Figure 12.5. You may achieve different results based on your random training–validation split of the original data.